THE ECONOMICS OF OBJECT-ORIENTED SOFTWARE

April 14, 1997

Capers Jones, Chairman
Software Productivity Research, Inc.
1 New England Executive Park
Burlington, MA 01803-5005

Phone	617 273 0140
FAX	617 273 5176
Email	Capers@SPR.com
CompuServe	75430,231

Abstract

In 1994 the author was commissioned by a European telecommunications company to perform an economic analysis of both procedural and object-oriented programming languages used for telecommunications software. It is appropriate to revisit this study in 1997 using several years of additional data.

The expansion of object-oriented (OO) programming languages and methodologies has brought with it a need to upgrade software estimating and measurement technologies. A number of measurement and estimating approaches were explored in the context of how well they support object-oriented approaches. The traditional "Lines of Code" or LOC metric appears to be hazardous for object-oriented languages. The Function Point metric provides useful information for OO projects and allows direct comparisons to traditional methods. The specialized metrics being developed explicitly for OO projects provide useful data in an OO context, but do not allow direct comparisons between OO projects and procedural projects.

Copyright ? 1994-1997 by Capers Jones, Chairman
Software Productivity Research. All rights reserved.

INTRODUCTION

In 1994 the author and his colleagues at Software Productivity Research were commissioned by a European telecommunications company to explore an interesting problem.

Many of this company's products were written in the CHILL programming language. CHILL is a fairly powerful third-generation procedural language developed specifically for telecommunications applications in the 1970's by the CCITT, an international telecommunications association.

Software engineers and managers within the company were interested in moving to object-oriented programming using C++ as the primary language. Studies had been carried out by the company to compare the productivity rates of CHILL and C++ for similar kinds of applications. These studies concluded that CHILL projects had higher productivity rates than C++ when measured with the productivity metric, "Lines of Code per staff month" or LOC per staff month.

We were asked to explore the results of these experiments, and either confirm or challenge the finding that CHILL was superior to C++. We were also asked to make recommendations about other possible languages such as Ada83, Ada95, C, PASCAL, or SMALLTALK.

As background information, we also examined the results of using macro-assembly language. All eight of these languages are either being used for telecommunications software, or were candidates for use by telecommunications software as in the case of Ada95 which was just being prepared for release.

Excerpts from the original study were published in Ed Yourdon's American Programmer magazine in October of 1994 (Jones 1994). This report extends the 1994 version by including two additional languages and releasing some additional information not in the prior version on schedules and release dates.

In this 1997 revision of the original 1994 analysis, two additional languages are included in the analysis: PL/I and Objective C. The PL/I language has been used for switching software applications and the construction of PBX switches since the 1970's. For example, several ITT switches during the 1970's were constructed in a PL/I variant called "Electronic Switching PL/I" or ESPL/I.

The Objective C language actually originated as a telecommunications language within the ITT Corporation under Dr. Tom Love at the ITT Programming Technology Center in Stratford, Connecticut. However the domestic ITT research facilities were closed after Alcatel bought ITT's telecommunications business, so the Objective C language was brought to the commercial market by Dr. Love and the Stepstone Corporation.

The results and methods of our exploration follow. The basic conclusion was that object-oriented languages did offer substantial economic productivity gains compared to third-generation procedural languages, but that these advantages were hidden when measured with the LOC metric.

However, object-oriented analysis and design is more troublesome and problematic. Both the Booch, Jacobsen, and Rumbaugh "flavors" of OO analysis and design had very steep learning curves and were often augmented or abandoned in order to complete projects that used them.

For a discussion of software failures and abandonments, refer to my book Patterns of Software Systems Failure and Success (Jones 1995)

The new Unified Modeling Language (UML) may eliminate some of these problems, but still has too little solid empirical data from use on real switching systems to judge its overall impact.

BASIS OF THE STUDY

The kind of project selected for this study was the software for a private branch exchange switch (PBX), similar to the kinds of private switching systems utilized by the larger hotels and office buildings.

The original data on CHILL was derived from the client's results, and data for the other programming languages were derived from other telecommunication companies who are among our clients. (The Ada95 results were originally modeled mathematically, since this language had no available compilers at the time of the original study. However since 1994 Ada 95 has expanded enough to have empirical results.)

To ensure consistent results, all versions were compared using the same sets of activities, and any activities that were unique for a particular project were removed. The data was normalized using our CHECKPOINT? measurement and estimation tool. This tool facilitates comparisons between different programming languages and different sets of activities, since it can highlight and mask activities that are not common among all projects included in the comparison. The full set of activities which we studied included more than 20, but a final presentation used consolidated data based on six major activities:

1. Requirements
2. Design
3. Coding
4. Integration and Testing
5. Customer Documentation
6. Management

The consolidation of data to six major activities was primarily to simplify presenting the results. The more granular data actually utilized included activity and task-level information. For example, the cost bucket labeled "integration and testing" really comprised information derived from integration, unit testing, new function testing, regression testing, stress and performance testing, and field testing.

For each testing stage, data was available on test case preparation, test case execution, and defect repair costs. However, the specific details of each testing step are irrelevant to an overall economic study. So long as the aggregation is based on the same sets of activities, this approach does not degrade the overall accuracy.

Since the original 1994 study concentrated primarily on object-oriented programming languages as opposed to object-oriented analysis and design, data from OO requirements and analysis were not explored in depth in the earlier report.

Other SPR clients and other SPR studies that did explore various OO analysis and design methods found that they had steep learning curves and did not benefit productivity in the near term. In fact the OO analysis and design approaches were abandoned or augmented by conventional analysis and design approaches in about 50% of the projects that attempted to use them initially.

The new unified modeling language (UML) which consolidates the methods of Booch, Rumbaugh, and Jacobsen is still too new to have accumulated much in the way of empirical observations in this version of the report, but a future version will add detailed information as it becomes available.

METRICS EVALUATED FOR THE STUDY

Since the main purpose of the study was to compare object-oriented languages and methods against older procedural languages and methods, it was obvious that we needed measurements and metrics that could handle both the old and the new. Since the study was aimed at the economic impact associated with entire projects and not just pure coding, it was obvious that the metric needed to be useful for measuring non-coding activities such as requirements, design, documentation, and the like. The metrics which were explored for this study included:

?? Physical lines of code, using the Software Engineering Institute (SEI) counting rules
?? Logical lines of code, using the Software Productivity Research (SPR) counting rules
?? Feature Points
?? Function Points (Version 3.4 for the 1994 study)
?? Function Points (Version 4.0 for the 1997 revised study)
?? MOOSE (Metrics for Object Oriented System Environments)

The SEI approach of using physical lines (Park, 92) was eliminated first, since the variability is both random and high for studies that span multiple programming languages. Individual programming styles can affect the count of physical lines by several hundred percent. When multiple programming languages are included, the variance can approach an order of magnitude.

The SPR approach of using logical statements gives more consistent results than a count of physical lines and reduces the variations due to individual programming styles. The usage of logical statements also facilitates a technique called "backfiring" or the direct conversion of LOC metrics into functional metrics. The use of logical statements using the SPR counting rules published in Applied Software Measurement (Jones 96) provides the basis

of the LOC counts shown later in this report. However, LOC in any form is not a good choice for dealing with non-coding activities such as document creation.

The Feature Point metric was originally developed for telecommunications software, and indeed is being used by some U.S. and European telecommunication companies such as Motorola for software measurement work. This metric is not as well known in Europe as the Function Point metric, however, so it was not used in the final report. (For those unfamiliar with the Feature Point metric, it adds sixth parameter which is a count of algorithms to the five parameters used by standard Function Point metrics. This metric is also described in my book Applied Software Measurement.)

The Function Point metric was selected for displaying the final results of this study, using a constant 1500 Function Points as the size of all eight versions. The Function Point metric is now the most widely utilized software metric in both the United States and some 20 other countries including much of Europe.

More data is now being published using this metric than any other, and the number of automated tools which facilitate counting Function Points is growing exponentially. Also, this metric was familiar to the European company for which the basic analysis was being performed.

For those unfamiliar with the Function Point metric, it is derived from the weighted counts of five external attributes of software applications:

1. Inputs
2. Outputs
3. Inquiries
4. Logical files
5. Interfaces

The counting rules used in the original study were those of Version 3.4 (IFPUG, 93). Refer to Dr. Brian Dreger's primer Function Point Analysis (Dreger, 89) for general background information on this metric. For those interested in the specific application of Function Points to OO projects, refer to Dr. Tom Love's book Object Lessons (Love, 93) for an interesting set of examples.)

In 1995 the International Function Point Users Group (IFPUG) issued a major revision to the function point counting rules, Version 4. Version 4 reduced the value of error messages in function point analysis, and as a result counts for many applications are about 20% lower using the Version 4 counting rules than using the previous Version 3.4 counting rules.

The first 1994 edition of this report obviously used the Version 3.4 rules since Version 4 were not available. This new 1997 edition adopts the Version 4 rules and hence there are some minor differences in numeric data between the two versions.

Since a key purpose of the study was to explore the economics of object-oriented programming languages, it was natural to consider using the new "metrics for object-oriented system environments" (MOOSE) developed by Dr. Chris Kemerer of MIT (Kemerer and Chidamber 1993).

The MOOSE metrics include a number of constructs that are only relevant to OO projects, such as Depth of the Inheritance Tree (DIT), Weighted Methods per Class (WMC), Coupling Between Objects (CBO), and several others.

However, the basic purpose of the study is a comparison that now includes 10 programming languages, of which six are older procedural languages. Unfortunately the MOOSE metrics do not lend themselves to cross-language comparisons between OO projects and procedural projects, and so had to be excluded.

In some ways, the Function Point metric resembles the "horsepower" metric. In the year 1783 James Watt tested a strong horse by having it lift a weight, and found that it could raise a 150 pound weight almost four feet in one second. He created the ad hoc empirical metric, "horsepower," as 550 foot pounds per second. The horsepower metric has been in continuous usage for more than 200 years and has served to measure steam engines, gasoline engines, diesel engines, electric motors, turbines, and even jet engines and nuclear power plants.

The Function Point metric also had an ad hoc empirical origin, and is also serving to measure types of projects that did not exist at the time of its original creation in the middle 1970's such as client-server software, object-oriented projects, and even multi-media applications.

Function point metrics originated as a unit for measuring size, but have also served effectively as a unit of measure for software quality, for software productivity, and even for exploring software value and return on investment. In all of these cases, function points are generally superior to the older lines of code (LOC) metric. Surprisingly, function points are also superior to some of the modern object-oriented (OO) metrics, which are often difficult to apply to quality or economic studies.

For those considering the selection of metrics for measuring software productivity and quality, eight practical criteria can be recommended:

Eight Suggested Criteria for Software Metrics Adoption

1) The metric should have a standard definition and be unambiguous.
2) The metric should not be biased and unsuited for large scale statistical studies.

3) The metric should have a formal user group and adequate published data.
4) The metric should be supported by tools and automation.
5) It is helpful to have conversion rules between the metric and other metrics.
6) The metric should deal with all software deliverables, and not just code.
7) The metric should support all kinds and types of software projects.
8) The metric should support all kinds and types of programming languages.

It is interesting that the Function Point metric is currently the only metric that meets all eight criteria.

The Feature Point metric lacks a formal user group and has comparatively few published results but otherwise is equivalent to Function Points and hence meets seven of the eight criteria.

The LOC metric is highly ambiguous, lacks a user group, is not suited for large scale statistical studies, is unsuited for measuring non-code deliverables such as documentation, and does not handle all kinds of programming languages such as visual languages or generators. In fact, the LOC metric does not really satisfy any of the criteria.

The MOOSE metrics are currently in evolution and may perhaps meet more criteria in the future. As this paper was written, the MOOSE metrics for object-oriented projects appeared unsuited for non-OO projects, do not deal with deliverables such as specifications or user documentation, and lack conversion rules to other metrics.

RESULTS OF THE ANALYSIS

The first topic of interest is the wide range in the volume of source code required to implement essentially the same application. Note that the Function Point total of the PBX project used for analysis was artificially held constant at 1500 Function Points across all 10 versions in this example.

In real-life, of course, the functionality varied among the projects utilized. Table 1 gives the volumes of source code and the number of source code statements needed to encode one Function Point for the 10 examples:

Table 1: Function Point and Source Code Sizes for 10 Versions of the Same Project (A PBX Switching System of 1,500 Function Points in Size)

Language	Size in Func. Pt.	Lang. Level	LOC per Func. PT.	Size in LOC
Assembly	1,500	1	250	375,000
C	1,500	3	127	190,500
CHILL	1,500	3	105	157,500
PASCAL	1,500	4	91	136,500
PL/I	1,500	4	80	120,000
Ada83	1,500	5	71	106,500
C++	1,500	6	55	82,500
Ada95	1,500	7	49	73,500
Objective C	1,500	11	29	43,500
Smalltalk	1,500	15	21	31,500
Average	1,500	6	88	131,700

The next topic of interest is the amount of effort required to develop the 10 examples of the project. Table 2 gives the effort for the six major activities and the overall quantity of effort expressed in terms ”staff months.” Note that the term ”staff month” is defined as a typical working month of about 22 business days, and includes the assumption that

project work occurs roughly 6 hours per day, or 132 work-hours per month. The data normalization feature of the CHECKPOINT? tool was used to make these assumptions constant, even though the actual data was collected from a number of companies in different countries, where the actual number of work hours per staff month varied.

Table 2: Staff Months of Effort for 10 Versions of the Same Software Project (A PBX Switching System of 1,500 Function Points in Size)

Language	Req. (Months)	Design (Months)	Code (Months)	Test (Months)	Doc. (Months)	Mgt. (Months)	TOTAL (Months)
Assembly	13.64	60.00	300.00	277.78	40.54	89.95	781.91
C	13.64	60.00	152.40	141.11	40.54	53.00	460.69
CHILL	13.64	60.00	116.67	116.67	40.54	45.18	392.69
PASCAL	13.64	60.00	101.11	101.11	40.54	41.13	357.53
PL/I	13.64	60.00	88.89	88.89	40.54	37.95	329.91
Ada83	13.64	60.00	76.07	78.89	40.54	34.99	304.13
C++	13.64	68.18	66.00	71.74	40.54	33.81	293.91
Ada95	13.64	68.18	52.50	63.91	40.54	31.04	269.81
Objective C	13.64	68.18	31.07	37.83	40.54	24.86	216.12
Smalltalk	13.64	68.18	22.50	27.39	40.54	22.39	194.64
Average	13.64	63.27	100.72	100.53	40.54	41.43	360.13

Note that since the data came from four companies each of which had varying accounting assumptions, different salary rates, different work month definitions, work pattern differences, and other complicating factors, the separate projects were run through the CHECKPOINT? tool and converted into standard work periods of 132 hours per month, costs of $10,000 per month.

None of the projects were exactly 1,500 function point in size, and the original sizes ranges from about 1,300 to 1,750 function points in size. Here too, the data normalization feature was used to make all 10 versions identical in factors that would conceal the underlying similarities of the examples.

It can readily be seen that the overall costs associated with coding and testing are much less significant for object-oriented languages than for procedural languages. However, the effort associated with initial requirements, design, and user documentation are comparatively inelastic and do not fluctuate in direct proportion to the volume of code required.

Note an interesting anomaly: The effort associated with analysis and design is higher for the object-oriented projects than for the procedural projects. This is due to the steep learning curve and general difficulties associated with the common ”flavors” or OO analysis and design: The Booch, Rumbaugh, and Jacobsen original methods.

Unfortunately, not enough data is yet available on the new UML or Unified Modeling Language to form a firm opinion, but this topic will be revisited in future versions of this report.

The most interesting results are associated with measuring the productivity rates of the 10 versions. Note how apparent productivity expressed using the metric ”Lines of Code per Staff Month” moves in the opposite direction from productivity expressed terms of ”Function Points per Staff Month.”

The date in Table 3 is derived from the last column of Table 2, or the total amount of effort devoted to the 10 projects. Table 3 gives the overall results using both LOC and Function Point metrics:

Table 3: Productivity Rates for 10 Versions of the Same Software Project (A PBX Switching system of 1,500 Function Points in Size)

Language	Effort	Funct. Pt.	Work Hrs.	LOC per	LOC per

	(Months)	per Staff Month	per Funct. Pt.	Staff Month	Staff Hour
Assembly	781.91	1.92	68.81	480	3.38
C	460.69	3.26	40.54	414	3.13
CHILL	392.69	3.82	34.56	401	3.04
PASCAL	357.53	4.20	31.46	382	2.89
PL/I	329.91	4.55	29.03	364	2.76
Ada83	304.13	4.93	26.76	350	2.65
C++	293.91	5.10	25.86	281	2.13
Ada95	269.81	5.56	23.74	272	2.06
Objective C	216.12	6.94	19.02	201	1.52
Smalltalk	194.64	7.71	17.13	162	1.23
Average	360.13	4.17	31.69	366	2.77

As can easily be seen, the LOC data does not match the assumptions of standard economics, and indeed moves in the opposite direction from real economic productivity. It has been known for many hundreds of years that when manufacturing costs have a high proportion of fixed costs and there is a reduction in the number of units produced, the cost per unit will go up.

The same logic is true for software. When a "Line of Code" is defined as the unit of production, and there is a migration from low-level procedural languages to object-oriented languages, the number of units that must be constructed declines. The costs of paper documents such as requirements and user manuals do not decline, and tend to act like fixed costs. This inevitably leads to an increase in the "Cost per LOC" for object-oriented projects, and a reduction in "LOC per staff month" when the paper-related activities are included in the measurements.

On the other hand, the Function Point metric is a synthetic metric totally divorced from the amount of code needed by the application. Therefore Function Point metrics can be used for economic studies involving multiple programming languages and object-oriented programming languages without bias or distorted results. The Function Point metric can also be applied to non-coding activities such as requirements, design, user documentation, integration, testing, and even project management.

In order to illustrate the hazards of LOC metrics without any ambiguity, table 3.1 simply ranks the 10 versions in descending order of productivity. As can be seen, the rankings are completely reversed between the function point list and the LOC list.

Table 3.1: Rankings of Productivity Levels Using Function Point Metrics and Lines of Code (LOC) Metrics

	Productivity Ranking Using Function Point Metrics		Productivity Ranking Using LOC Metrics
1	Smalltalk	1	Assembly
2	Objective C	2	C
3	Ada95	3	CHILL
4	C++	4	PASCAL
5	Ada83	5	PL/I
6	PL/I	6	Ada83
7	PASCAL	7	C++
8	CHILL	8	Ada95

9	C	9	Objective C
10	Assembly	10	Smalltalk

When using the standard economic definition of productivity, which is ”*goods or services produced per unit of labor or expense*” it can be seen that the function point ranking matches economic productivity assumptions.

The function point ranking matches economic assumptions because the versions with the lowest amounts of both effort and costs have the highest function point productivity rates and the lowest costs per function point rates.

The LOC rankings, on the other hand, are the exact reversal of real economic productivity rates. This is the key reason why usage of the LOC metric is viewed as ”professional malpractice” when it is used for cross-language productivity or quality comparisons involving both high-level and low-level programming languages.

When the costs of the 10 versions are considered, the hazards of using ”lines of code” as a normalizing metric become even more obvious. Table 4 shows the total costs of the ten versions, and then normalizes the data using both ”Cost per line of code” and ”cost per Function Point.”

Note that telecommunications companies, as a class, tend to be on the high side in terms of burden rates and compensation and typically average from $9,000 to more than $12,000 per staff month for their fully burdened staff compensation rates.

The 10 examples shown in Table 4 are all arbitrarily assigned a fully burdened monthly salary rate of $10,000. Following is Table 4:

Table 4: Cost of Development for 10 Versions of the Same Software Project (A PBX Switching System of 1,500 Function Points in Size)

Language	Effort (Months)	Burdened Salary (Months)	Burdened Costs	Burdened Cost per Funct. Pt.	Burdened Cost per LOC
Assembly	781.91	$10,000	$7,819,088	$5,212.73	$20.85
C	460.69	$10,000	$4,606,875	$3,071.25	$24.18
CHILL	392.69	$10,000	$3,926,866	$2,617.91	$24.93
PASCAL	357.53	$10,000	$3,575,310	$2,383.54	$26.19
PL/I	329.91	$10,000	$3,299,088	$2,199.39	$27.49
Ada83	304.13	$10,000	$3,041,251	$2,027.50	$28.56
C++	293.91	$10,000	$2,939,106	$1,959.40	$35.63
Ada95	269.81	$10,000	$2,698,121	$1,798.75	$36.71
Objective C	216.12	$10,000	$2,161,195	$1,440.80	$49.68
Smalltalk	194.64	$10,000	$1,946,425	$1,297.62	$61.79
Average	360.13	$10,000	$3,601,332	$2,400.89	$27.34

Note the paradoxical increase in ”cost per line of code” for the more powerful languages, at the same time that both total costs and cost per Function Point decline. Using a metric such as LOC that moves in the opposite direction as economic productivity improves is a very hazardous practice.

Indeed, the LOC metric is so hazardous for cross-language comparisons that a strong case can be made that using ”lines of code” for normalization of data involving multiple or different languages should be considered an example of professional malpractice.

The phrase *"professional malpractice"* implies that a trained knowledge worker did something that was hazardous and unsafe, and that the level of training and prudence required to join the profession should have been enough to avoid the unsafe practice. Since it is obvious that the "lines of code" metric does not move in the same direction as economic productivity, and indeed moves in the opposite direction, it is a reasonable assertion that misuse of lines of code metrics should be viewed as professional malpractice if a report or published data caused some damage or harm.

One of the severe problems of the software industry has been the inability to perform economic analysis of the impact of various tools, methods, or programming languages. It can be stated that the "lines of code" or LOC metric has been a significant barrier that has slowed down the evolution of software engineering, since it has blinded researchers and prevented proper exploration of software engineering factors.

In table 5, we show some information that was not part of the original 1994 study: the development schedules and the total staffing required to build the software applications. Also shown is another factor which was not in the 1994 study: The year in which the projects were actually deployed.

Recall that although this study was performed for one telecommunications company, the data was actually derived from four telecommunications companies and the projects span a chronological time period from 1979 through 1996. Unfortunately for specific projects such as "A PBX switching system of 1,500 function points in size" there are not enough such projects in any given year to perform a meaningful analysis.

Table 5: Overall Staffing, Effort, and Schedule for 10 Versions of the Same Project (A PBX Switching System of 1,500 Function Points in Size)

Language	Effort (Months)	Staff Total	Schedule (Months)	Year of Release
Assembly	781.91	17.65	44	1979
C	460.69	13.64	34	1984
CHILL	392.69	12.50	31	1985
PASCAL	357.53	12.00	30	1987
PL/I	329.91	11.54	29	1989
Ada83	304.13	10.71	28	1994
C++	293.91	10.34	28	1994
Ada95	269.81	10.00	27	1996
Objective C	216.12	9.09	24	1995
Smalltalk	194.64	9.09	21	1994
Average	360.13	11.66	30	1990

The use of software for switching applications is now a technology that is more than 25 years old. Although the productivity, staffing, and schedule results of the object-oriented versions are significantly better than for the procedural version, the OO versions are all fairly recent and were created by software teams with a great deal of experience in automated switching applications.

By contrast, the 1979 assembly language version was built by a team to whom automated switching software was a novel exercise, rather than a common kind of software project. Some of the differences in the overall results are not necessarily due to the virtues of object-oriented approaches themselves, but rather to the switching software experience levels of the teams who are using the object-oriented approaches.

The age of the software applications also seems to play a major part in the volume of reusable materials utilized. One of the major claims of the OO paradigm is that increased volumes of reusable materials will be available. However these claims are mostly subjective assertions that are not yet fully supported by empirical data.

Software reuse is a multi-faceted topic which involves much more than reusable source code. An really effective software reuse program can include a dozen artifacts, and at a very minimum should include five critical artifacts:

1. Reusable requirements
2. Reusable designs and specifications
3. Reusable source code
4. Reusable test cases
5. Reusable user documentation

Unfortunately much of the literature on software reuse, including the OO literature on software reuse, has concentrated exclusively on code reuse. For example, the OO literature is almost silent on reuse of user documentation and fairly sparse on the reuse of requirements, design, and test materials.

Hopefully the unified modeling language will increase the emphasis on reusable design, but the OO literature is still sparse on the reuse of many other software artifacts.

Table 6 illustrates the approximate percentages of reuse in the 10 projects. The data has a very high margin of error and is derived primarily from the subjective views of participants in the projects, often reconstructed from memory. Here too, additional research is clearly indicated.

Table 6: Approximate Percentages of Reuse for 10 Versions of the Same Project (A PBX Switching System of 1,500 Function Points)

Language	Reusable Require.	Reusable Design	Reusable Code	Reusable Tests	Reusable Docum.	Reuse Average
Assembly	10.00%	5.00%	5.00%	10.00%	10.00%	8.00%
C	10.00%	7.00%	10.00%	10.00%	10.00%	9.40%
CHILL	15.00%	10.00%	10.00%	10.00%	20.00%	13.00%
PASCAL	15.00%	15.00%	12.00%	10.00%	20.00%	14.40%
PL/I	15.00%	15.00%	15.00%	10.00%	20.00%	15.00%
Ada83	15.00%	20.00%	25.00%	15.00%	25.00%	20.00%
C++	40.00%	30.00%	35.00%	15.00%	25.00%	29.00%
Ada95	50.00%	35.00%	40.00%	20.00%	25.00%	34.00%
Objective C	50.00%	40.00%	55.00%	20.00%	35.00%	40.00%
Smalltalk	50.00%	40.00%	55.00%	20.00%	35.00%	40.00%
Average	27.00%	21.70%	26.20%	14.00%	22.50%	22.28%

Although the OO projects do have a higher volume of reuse than the procedural projects, it is not clear if the reuse is due to the OO approach itself, or to the fact that the OO projects are all products of the mid 1990's while some of the procedural projects are products of the 1970's and 1980's when reusable materials were not emphasized and unavailable.

In terms of requirements reuse and user documentation reuse, it is obvious that the OO approach itself has nothing to do with the situation whatsoever. The high levels of reuse are because the teams and companies had already built several earlier generations of PBX switching systems, so the OO versions were simply being constructed on top of a substantial knowledge base of well-known features and deliverables from the earlier versions.

SOFTWARE QUALITY ANALYSIS

Another interesting aspect of the study was the exploration of quality results, using ”defect potentials” and ”defect removal efficiency” as the primary metrics. The defect potential of a software project is the sum of the errors found in requirements, design, code, user documentation, and bad fixes (”bad fixes” are. secondary errors introduced when repairing prior defects).

The defect removal efficiency of a project is the total percentage of defects eliminated prior to delivery of software to its intended clients. Defect removal efficiency is normally calculated on the anniversary of the delivery of software. For example, if the development team found a total of 900 bugs in a software project and the users reported a total of 100 bugs in the first year of usage, then the defect removal efficiency for that project would obviously be calculated to be 90% when both sets of bug reports are summed and the ratio of pre-release defects is calculated. Table 7 summarizes the defect potentials of the 10 projects:

Table 7: Defect Potentials for 10 Versions of the Same Software Project (A PBX Switching System of 1,500 Function Points in Size)

Language	Req. Defects	Design Defects	Coding Defects	Doc. Defects	Bad Fix Defects	TOTAL DEFECTS
Assembly	1,500	1,875	7,500	900	1,060	12,835
C	1,500	1,875	3,810	900	728	8,813
CHILL	1,500	1,875	3,150	900	668	8,093
PASCAL	1,500	1,875	2,730	900	630	7,635
PL/I	1,500	1,875	2,400	900	601	7,276
Ada83	1,500	1,875	2,130	900	576	6,981
C++	1,500	2,025	1,650	900	547	6,622
Ada95	1,500	2,025	1,470	900	531	6,426
Objective C	1,500	2,025	870	900	477	5,772
Smalltalk	1,500	2,025	630	900	455	5,510
Average	1,500	1,935	2,634	900	627	7,596

Note that the columns of requirements defects and documentation defects were held constant using the data normalization feature of our CHECKPOINT? tool. The older projects did not record this value, and since all of the projects did the same thing, it was reasonable to use a constant value across all 10 versions of this similar project.

The large volume of coding defects for the assembly language and C language versions of the projects should come as no surprise to programmers who have used these rather taxing languages for complex real-time software such as switching systems.

Table 8 shows the defect removal efficiency levels, or the number of defects eliminated before deployment of the software via design inspections, code inspections, and testing. Note that telecommunications software, as a class, is much better than U.S. averages in terms of defect removal efficiency levels due in part to the wide-spread usage of formal design reviews, formal code inspections, testing specialists, and the use of state of the art defect tracking and test automation tools.

Table 8 uses the metric ”defects per KLOC” rather than ”defects per LOC.” The term ”KLOC” refers to units of 1000 lines of code, and is preferred for quality metrics because defects per LOC would usually require three zero’s before encountering a value and it is hard to visualize quality with so many decimal places.

Table 8: Delivered Defects for 10 Versions of the Same Software Project

(A PBX Switching System of 1,500 Function Points in Size)

Language	Total Defects	Defect Removal Efficiency	Delivered Defects	Delivered Defects per Funct. Pt.	Delivered Defects per KLOC
Assembly	12,835	91.00%	1,155	0.77	3.08
C	8,813	92.00%	705	0.47	3.70
CHILL	8,093	93.00%	567	0.38	3.60
PASCAL	7,635	94.00%	458	0.31	3.36
PL/I	7,276	94.00%	437	0.29	3.64
Ada83	6,981	95.00%	349	0.23	3.28
C++	6,622	93.00%	464	0.31	5.62
Ada95	6,426	96.00%	257	0.17	3.50
Objective C	5,772	96.00%	231	0.15	5.31
Smalltalk	5,510	96.00%	220	0.15	7.00
Average	7,580	94.00%	455	0.30	3.45

For a recent survey of the state of the art of software defect removal, refer to my book Software Quality - Analysis and Guidelines for Success (Jones 1997).

It is immediately apparent that the metric "Defects per KLOC" is not suitable for measuring defect volumes that include errors made in requirements, design, and user documentation as the data in Table 8 does. For quality, just as for productivity, the LOC metric reverses the real situation and yields incorrect conclusions.

Although coding defects dominate in the older assembly and C projects, design defects dominate in the newer OO projects. These non-coding defects amount to more than 50% of the total defect load for the more recent projects, and may amount to more than 80% of the defect load for object-oriented projects.

The metric "Defects per Function Point" is a much better choice for dealing with all of the possible sources of software error, and especially so for OO projects.

There is a subtle problem with measuring quality in a modern OO environment. As might be expected, the defect potentials for the object-oriented versions are lower than for the procedural versions. If you examine pure coding defects, it is surprising is that the overall defect removal efficiency rates decline for projects using OO languages. This is a counter-intuitive phenomenon, but one which is supported by substantial empirical data.

The reason for this seeming quality paradox is actually very logical once it is understood. Coding defects are easier to get rid of than any other category. Defect removal efficiency for pure coding errors is usually higher than 95%. For requirements and design errors, defect removal is much tougher, and averages less than 80%. When OO programming languages are used, there is a significant reduction in coding errors as a result of the defect prevention aspects of OO languages. This means that the troublesome requirements and design errors constitute the bulk of the defects encountered on OO projects, and they are very hard to eliminate.

However, the telecommunications community has generally been aware of this problem and has augmented their testing stages by a series of pre-test design and code inspections. Also, for the more recent projects the set of requirements and design errors are reduced because these projects are no longer novel, but are being constructed as simply the most recent links in a growing chain of similar projects.

For additional information on defect potentials and software defect removal, refer to my books Assessment and Control of Software Risks (Jones, 94).

CONCLUSIONS OF THE STUDY

The most obvious conclusions from the study were these four:

1) Object-oriented programming languages appear to benefit both software productivity and software quality.

2) There is not yet any solid data which indicates that object-oriented analysis and design methods benefit either software productivity or quality, although there is insufficient information about the unified modeling language (UML) to draw a conclusion.

3) Neither the quality nor productivity benefits of object-oriented programming can be directly measured using ”lines of code” or LOC metrics.

4) The synthetic Function Point metric appears to be able to bridge the gap between procedural and OO projects and can express results in a fashion that matches standard economic assumptions about productivity.

The Function Point metric is also a useful tool for exploring software quality, when used in conjunction with other metrics such as defect removal efficiency rates.

A somewhat surprising result of the original study was that productivity and quality cannot be compared between OO projects and non-OO projects using specialized object-oriented metrics such as MOOSE as they were currently described. It is possible to use OO metrics for comparisons within the OO paradigm, but not outside the OO paradigm.

Some less obvious conclusions were also reached regarding the completeness of the object-oriented paradigm itself. Because coding productivity tends to increase more rapidly than the productivity associated with any other activity with OO projects, it would be unsafe to use only coding as the basis for estimating complete OO projects.

This finding also has a corollary, which is that the OO paradigm is not yet fully developed for the paper-related activities dealing with requirements, design, and user documentation. The new Unified Modeling Language (UML) may have a significant impact on OO analysis and design, but it is premature to know for sure as this report is written.

It should be pointed out that the OO literature has lagged in attempting to quantify the costs of the paper deliverables associated with software. A more subtle observation appears to be that both inspection methods and testing methods for OO projects is also in need of further research and development.

Neither the first nor second versions of this study are completely rigorous and do not attempt to control all sources of error. There may be a substantial margin of error with the results, but hopefully the major conclusions are reasonable. In any case, a single study is not sufficient to draw global conclusions so the data should be viewed as provisional.

Note also that the telecommunications industry is usually below U.S. averages in software productivity rates, although one of the top five U.S. industries in terms of software quality levels. The high complexity levels and high performance and reliability requirements of telecommunications software invokes a need for very careful development practices, and very thorough quality control.

In addition, telecommunications software ranks number two out of all U.S. industries in terms of the total volume of text documents and paper materials created for software projects (military software ranks number one in the paperwork production factor). Telecommunications companies are also above U.S. averages in terms of burdened salary rates. Therefore the data shown here should not be used for general estimating purposes.

REFERENCES

Dreger, Brian J.; Function Point Analysis; Prentice Hall, Englewood Cliffs, NJ; 1989; 185 pages.

Garmus, David (Editor); IFPUG Counting Practices Manual, Release 3.4; International Function Point Users Group (IFPUG); Westerville, OH; 1993.

Jones, Capers: "The Economics of Object-Oriented Software; American Programmer Magazine, October 1994; 29-35.

Jones, Capers: Assessment and Control of Software Risks; Prentice Hall, Englewood Cliffs, NJ; 1994; 619 pages.

Jones, Capers; Patterns of Software Systems Failure and Success; International Thomson Computer Press, Boston, 1995; 292 pages.

Jones, Capers; Applied Software Measurement - Revised 2n edition; McGraw-Hill, New York; 1996; 618 pages.

Jones, Capers; Software Quality - Analysis and Guidelines for Success; International Thomson Computer Press, Boston; 1997; 472 pages.

Kemerer, Chris E. and Chidamber, S.R; MOOSE: Metrics for Object Oriented System Environments; Proceedings of the ASM 93 Conference.

Love, Tom; Object Lessons -- Lessons Learned in Object-Oriented Development Projects; SIGS Books, New York; 1993; 266 pages.

Park, Robert E.: SEI-92-TR-20: Software Size Measurement: A Framework for Counting Software Source Statements; Software Engineering Institute, Pittsburgh, PA; 1992; 220 pages.

ADDITIONAL READINGS

Chidamber S.R. and Kemerer, Chris F.: Toward a Metrics Suite for Object Oriented Design; OOPSLA 1991; pp 197-211.

Chidamber, S.R., Darcy, David P., and Kemerer, Chris F.; Managerial Use of Object-Oriented Software Metrics; Graduate School of Business, University of Pittsburgh; Working paper #750; February 1997.

Garmus, David (Editor); IFPUG Counting Practices Manual, Release 4.0; International Function Point Users Group (IFPUG); Westerville, OH; April 1994.

Henry, Sallie et al; Software Metrics Generation Tool for the Object Oriented Paradigm; Proceedings of the ASM 93 Conference.

Jones, Capers; Critical Problems in Software Measurement; IS Management Group, Carlsbad, CA; 1993; 104 pages.

Jones, Capers; New Directions in Software Management; IS Management Group, Carlsbad, CA; 1994; 150 pages.

Jones, Capers; Patterns of Software System Failure and Success; International Thomson Press, Boston, MA; 1995; 292 pages.

Kemerer, Chris F.: Reliability of Function Point Measurement: A Field Experiment; MIT Sloan School Working Paper 3192-90-MSA; January 1991.

Pfleeger, Shari Lawrence, Carleton, Anita et al; CMU/SEI-92-TR-19: Software Measurement for DoD Systems: Recommendations for Initial Core Measures; Software Engineering Institute, Pittsburgh, PA; 1992.

Rozum, James A.; SEI-92-TR-11; Software Measurement Concepts for Acquisition Program Mangers; Software Engineering Institute, Pittsburgh, PA; 1992.